Al Sham Private University
Faculty of Informatics Engineering

جامعة الشام الخاصة
كلية الهندسة المعلوماتية

نظم معلومات موزعة
Distributed Information Systems

Lecture 4: Remote invocation

Request-reply protocols & Remote procedure call (RPC)

اعداد: أ. غاندي هسام

Introduction

- This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system.
- **Request-reply protocols** represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing.
- The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the remote procedure call, or **RPC**, model).
- In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of remote method invocation (**RMI**).

Request-reply protocols

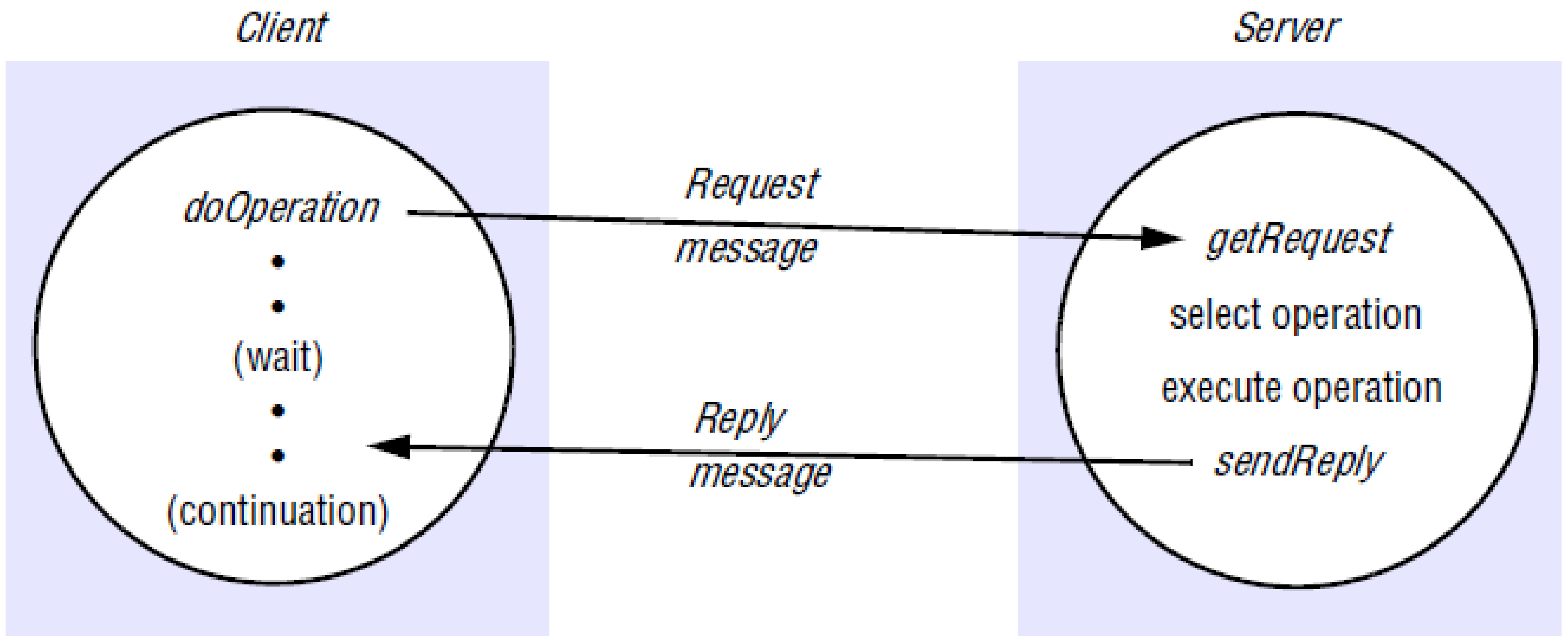
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- It can also be reliable because the reply from the server is effectively an acknowledgement to the client.
- The client-server exchanges are described in the following paragraphs in terms of the send and receive operations in the Java API for UDP datagrams, although many current implementations use TCP streams.

❑ The request-reply protocol:

- The protocol we describe here is based on a three of communication primitives, *doOperation*, *getRequest* and *sendReply*

- The *doOperation* method is used by clients to invoke remote operations.

Request-reply communication



HTTP: An example of a request-reply protocol

- used by web browser clients to make requests to web servers and to receive replies from them.
- web servers manage resources implemented in different ways:
  - as data – for example the text of an HTML page, an image or the class of an applet;
  - as a program – for example, servlets or PHP or Python programs that run on the web server.
- Client requests specify a URL that includes the DNS hostname of a web server and an optional port number on the web server as well as the identifier of a resource on that server.
- HTTP is a protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results, and the rules for representing (**marshalling**) them in the messages.

- HTTP is implemented over TCP. In the original version of the protocol, each client/server interaction consisted of the following steps:
  - The client requests and the server accepts a connection at the default server port or at a port specified in the URL.
  - The client sends a request message to the server.
  - The server sends a reply message to the client.
  - The connection is closed.
- Requests and replies are marshalled into messages as ASCII text strings, but resources can be represented as byte sequences and may be compressed.

HTTP methods

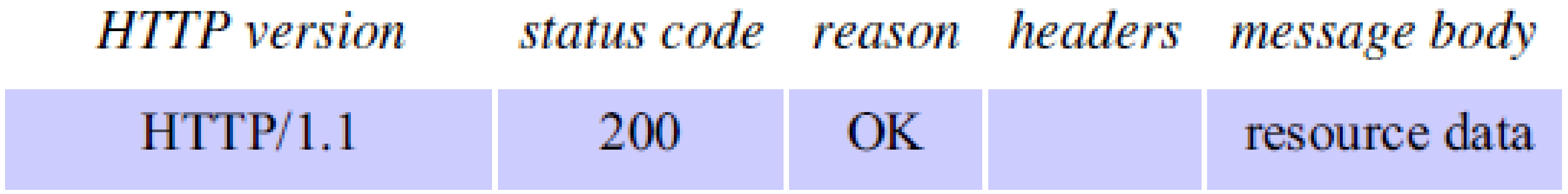
- Each client request specifies the name of a method to be applied to a resource at the server and the URL of that resource. The reply reports on the status of the request.
- Requests and replies may also contain resource data, the contents of a form or the output of a program resource run on the web server.
- The methods include the following:

➢ **GET**: Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified by that URL. If the URL refers to a program, then the web server runs the program and returns its output to the client.

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

- **HEAD**: This request is identical to GET, but it does not return any data. However, it does return all the information about the data, such as the time of last modification, its type or its size.
- **POST**: Specifies the URL of a resource (for example a program) that can deal with the data supplied in the body of the request. The processing carried out on the data depends on the function of the program specified in the URL.
- **PUT**: Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.
- ***DELETE***: The server deletes the resource identified by the given URL.

- A Reply message specifies the protocol version, a status code and ‘reason’, some headers and an optional message body
- The status code and reason provide a report on the server’s success



HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

- The header fields are used to pass additional information about the server or access to the resource For example, if the request requires authentication.

Remote procedure call

Programming with interfaces

- Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another.
- Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module.
- In order to control the possible interactions between modules, an explicit ***interface*** is defined for each module.
- The interface of a module specifies the procedures and the variables that can be accessed from other modules.
- So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems

- In a distributed program, the modules can run in separate processes. In the client-server model, in particular, each server provides a set of procedures that are available for use by clients.
- For example, a file server would provide procedures for reading and writing files.
- The term ***service interface*** is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.
- Programmers also do not need to know the programming language or underlying platform used to implement the service (an important step towards managing heterogeneity in distributed systems).

Interface definition languages

- An **RPC** mechanism can be integrated with a particular programming language if it includes an suitable notation for defining interfaces.
- This approach is useful when all the parts of a distributed application can be written in the same language for local and remote invocation.
- Interface definition languages (**IDL**s) are designed to allow procedures implemented in different languages to invoke one another.
- An IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified.
- The concept of an IDL was initially developed for RPC systems but applies equally to **RMI** and also **web services**.

Transparency

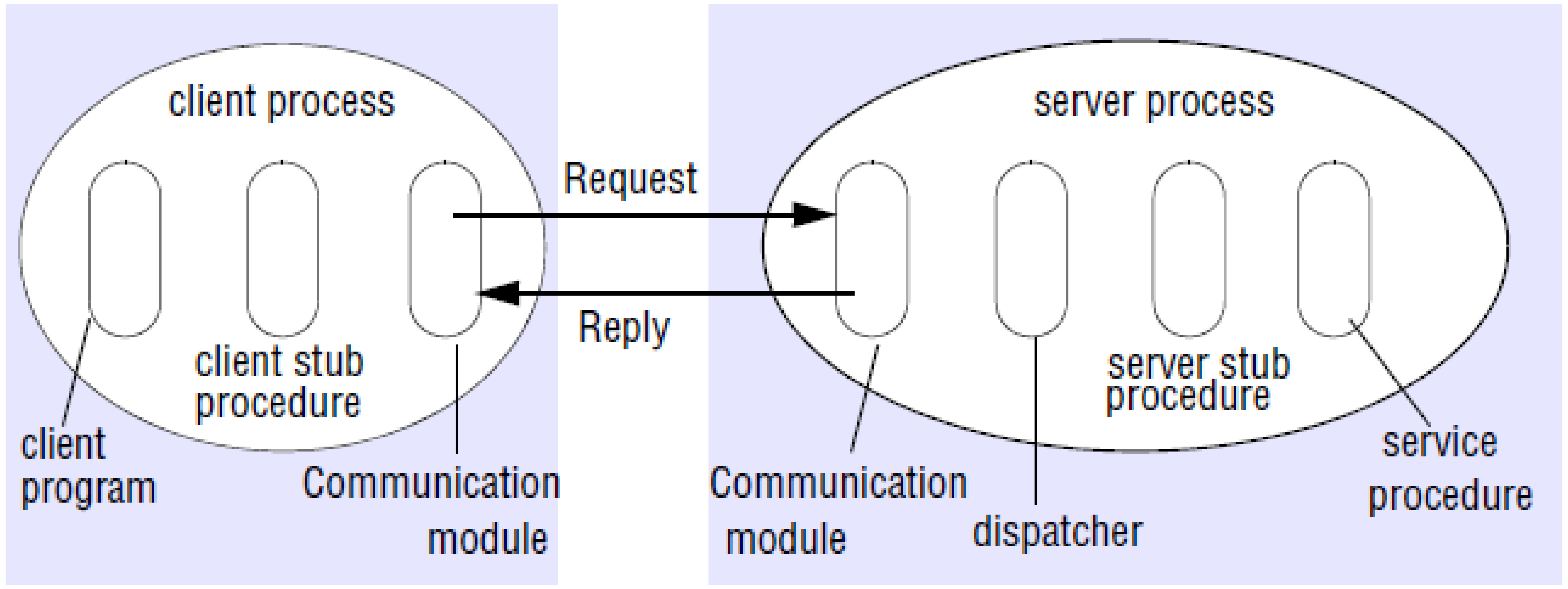
- The originators of RPC [1984], aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call.
- All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call.
- Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.
- RPC try hard to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way.
- **Middleware** can also offer additional levels of transparency to RPC.

Implementation of RPC

- The client that accesses a service includes one ***stub procedure*** for each procedure in the service interface.
- The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server.
- When the reply message arrives, it unmarshals the results.
- The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface.
- The dispatcher selects one of the **server stub procedures** according to the procedure identifier in the request message.

- The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message.
- The service procedures implement the procedures in the service interface.
- The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.
- RPC is generally implemented over a request-reply protocol.

Role of client and server stub procedures in RPC



Case study: *XML-RPC*

- we're going to create a server that uses Java to process XML-RPC messages, and create a Java client to call procedures on that server.
- The Java side of the conversation uses the Apache XML Project's Apache XML-RPC.
- Download jar package from https://drive.google.com/open?id=10IoGC0b1DFPBA18bS19HmRUiNzMC5NnP
- Use **NetBeans** IDE to create tow projects (RPC client and RPC server).
- Put all the .jar file in appropriate path and let us create one client and one small XML-RPC server using JAVA.

XML-RPC Client

```
import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient {
   public static void main (String [] args) {

      try {
         XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");
         Vector params = new Vector();

         params.addElement(new Integer(17));
         params.addElement(new Integer(13));

         Object result = server.execute("sample.sum", params);

         int sum = ((Integer) result).intValue();
         System.out.println("The sum is: "+ sum);

      } catch (Exception exception) {
         System.err.println("JavaClient: " + exception);
      }
   }
}
```

XML-RPC Server

```
import org.apache.xmlrpc.*;

public class JavaServer {

   public Integer sum(int x, int y){
      return new Integer(x+y);
   }

   public static void main (String [] args){

      try {

         System.out.println("Attempting to start XML-RPC Server...");

         WebServer server = new WebServer(80);
         server.addHandler("sample", new JavaServer());
         server.start();

         System.out.println("Started successfully.");
         System.out.println("Accepting requests. (Halt program to stop.)");

      } catch (Exception exception){
         System.err.println("JavaServer: " + exception);
      }
   }
}
```

XML Request format from Client

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
   <methodName>sample.sum</methodName>
   <params>
      <param>
         <value><int>17</int></value>
      </param>

      <param>
         <value><int>13</int></value>
      </param>
   </params>
</methodCall>
```

XML Response format from Server

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
   <params>
      <param>
         <value><int>30</int></value>
      </param>
   </params>
</methodResponse>
```

End of Lecture 4